Introducción

En este problema se nos presenta con una cantidad de oficinas N, las cuales están todas conectadas y están representadas como puntos en un eje cartesiano, se desea proveer de internet a las mismas.

A su vez se nos provee una cantidad W de módems que pueden ser instalados en cualquier oficina. Como sabemos que $W < N$, no podemos poner un módem por oficina y vamos a necesitar conectar las mismas mediante cables con una distancia determinada. También tenemos una distancia R la cual nos indica hasta qué distancia nos conviene usar cable UTP, la cual una vez superada, nos conviene usar fibra óptica. Los valores de cada tipo de cable van a estar dados por U (para los cables UTP) y V (para la fibra óptica).

Nos piden encontrar el costo mínimo necesario correspondiente a cada tipo de cable de forma que todas las oficinas tengan acceso a internet.

Dadas las características del problema podemos modelar el mismo como un grafo pesado completo, donde cada vértice corresponde a una oficina, y el peso de cada arista representa el precio del cable que conecta ambas oficinas.

Algoritmo

Se nos pide implementar un algoritmo eficiente basado en el algoritmo de Kruskal que resuelva el problema.
Para eso hacemos lo siguiente:

1. Inicializamos un vector con las `oficinas`, lo cual es $O(N)$.

2. Calculamos los pesos correspondientes a cada arista (y también su tipo de cable correspondiente) y los almacenamos en el vector `pesos`.
   Para esto tenemos que iterar sobre el vector de oficinas, lo cual es $O(N)$ e ir calculando la distancia entre todas, que es $O(N^2)$, fijarnos en $O(1)$ a qué tipo de cable corresponde, e insertarlo en el vector `pesos`, que también es $O(1)$. Todo esto es $O(N^2)$.

3. Una vez que tenemos esos dos vectores procedemos a construir el grafo. Primero creamos un vector de aristas de tamaño N, luego iteramos sobre el vector de oficinas en $O(N)$ y vamos creando las aristas que unen a cada nodo con todos los demás (recordemos que estamos trabajando con un grafo completo) y las insertamos. Esto es $O(N^2)$.
   Luego iteramos sobre el vector de `pesos`, lo cual es $O(N)$ y por cada iteración i le asignamos el peso correspondiente a la arista i. La complejidad total de hacer todo esto será $O(N^2)$.

4. Corremos una versión modificada de Kruskal sobre el grafo, basada en la implementación dada en la clase 9. Esta nueva versión lo que hace es interrumpir la ejecución una vez que Kruskal nos generó una determinada cantidad de bosques (CCs) generadores mínimos. En este caso vamos a querer tener W componentes conexas, o sea una por cada nodo donde pongo un módem. A su vez por cada iteración de Kruskal nos guardamos las aristas que agrega DSU cuando hace la unión en un vector `bosque` de tamaño N (o sea nos guardamos las aristas que conforman los bosques).
   Como usamos la versión provista en la clase 9 la cual tiene complejidad $O(E \log E)$, donde E es la cantidad de aristas del grafo y guardarnos las aristas es $O(1)$ y sabemos que $W < N$, entonces en particular podemos acotar la complejidad por $O(E \log E)$.

5. Una vez que Kruskal nos genera W componentes conexas y nos guarda las aristas que forman cada una, recorremos el vector `bosque` que las contiene y vamos sumando los precios de cada tipo de cable según corresponda. Esto va a ser $O(N)$.

6. Finalmente devolvemos mediante la salida estándar los precios acumulados de los dos tipos de cable, respetando el formato pedido.

Complejidad final: $O(N^2)$

<u>Justificación</u>
Podemos asegurar que el algoritmo es correcto gracias a lo que nos dice el invariante del algoritmo de Kruskal. Este invariante nos asegura que por cada iteración del algoritmo nos va generando un bosque de tamaño i (siendo i el número de iteraciones) con peso mínimo. Entonces nosotros lo que hacemos es iterar hasta que nos queden W componentes conexas, o sea ejecutamos Kruskal hasta que $N - i = W$ (esto representaría el hecho de colocar un módem por cada componente conexa).

Tener exactamente W componentes conexas con peso mínimo, nos da la solución al problema, porque para que una oficina tenga internet va a tener que tener un módem o estar conectada a alguna que tenga un módem mediante un cable. A nosotros nos piden que el costo de cada tipo de cable sea mínimo y esto lo logramos teniendo exactamente W componentes conexas, cada una con un módem, y donde las oficinas están todas conectadas a la que tiene módem con un camino de peso mínimo.

Sabemos por el invariante de Kruskal que los bosques van a ser mínimos, lo que significa que las aristas que componen cada uno, van a ser de peso mínimo, y como el peso representa el precio del cable que une dos oficinas, nos basta con ir sumando los pesos de las aristas de los bosques, que se forman por cada iteración de Kruskal a un acumulador, que represente el precio de su tipo de cable correspondiente.
Entonces, por todo lo anteriormente dicho, el valor final de cada variable donde acumulamos el costo de los tipos de cable va a ser mínimo, y todas las oficinas tendrán internet.

Experimentación

Para la experimentación de este problema nos piden analizar de forma empírica el comportamiento de dos versiones distintas de Kruskal, una optimizada y la otra no. Nosotros decidimos que para la versión sin optimizar, realizamos DSU sin hacer *union by rank* ni *path compression*. Teniendo en cuenta esto sabemos que en la implementación de DSU vista en clase, las operaciones *find* y *union* tienen una complejidad temporal de $O(\alpha(n))$, mientras que en la versión naive sin optimizaciones, dichas operaciones tienen costo $O(n)$.
Así, Kruskal con DSU sin optimizar, tiene complejidad de $O(E \log E + V^2)$, y Kruskal con DSU con las optimizaciones descritas previamente, tiene complejidad $O(E \log E)$.

Nuestra hipótesis es que la versión sin optimizar, va a ser menos eficiente que la optimizada si las oficinas están colocadas en línea recta, y la diferencia va a ser más marcada, a favor de la versión optimizada, a medida que aumente la cantidad de nodos. Para ver esto generamos distintos sets de peor caso para la versión naive de DSU, cumpliendo con las restricciones del enunciado, menos la que se hace sobre N. Nosotros acotamos $1 \leq N \leq 10000$ con la intención de que se noten más las diferencias de performance entre ambas implementaciones. También sólo tomamos el tiempo de la ejecución de Kruskal y no de todo el algoritmo (lo cual incluye leer el input), porque sólo nos interesa la performance de esa parte y no queremos "ruido" en nuestras mediciones.

Todos los test fueron ejecutados 10 veces en la misma PC, bajo las mismas condiciones, promediando los resultados.
Especificaciones de la PC:

- CPU: Ryzen 7 3800x (8 cores, 16 threads, 4.2 GHz)
- RAM: 16 GB
- SO: Windows 11

Los datos obtenidos para los distintos casos de test, así como el script utilizado para generar los mismos se encuentran en la carpeta correspondiente a este ejercicio.

Antes de ver los resultados de la experimentación, vamos a describir las complejidades de las diferentes versiones de DSU.
Primero analicemos el caso de la versión naive. En esta, la función de *find* termina siendo lineal y depende de la altura del árbol que construye DSU, ya que iteramos por los nodos hasta llegar a la raíz del árbol, que como máximo tiene altura $O(n)$. Por lo tanto su complejidad de peor caso termina siendo $O(n)$, donde n es la cantidad de nodos.
Union también termina siendo lineal ya que depende de *find*, y todas las otras operaciones que hace son $O(1)$. Entonces la complejidad de peor caso va a ser $O(n)$.

Por otro lado, y ya haciendo referencia a la versión optimizada de DSU, al aplicar la optimización *path compression*, lo que conseguimos es acortar la altura del árbol sobre el que

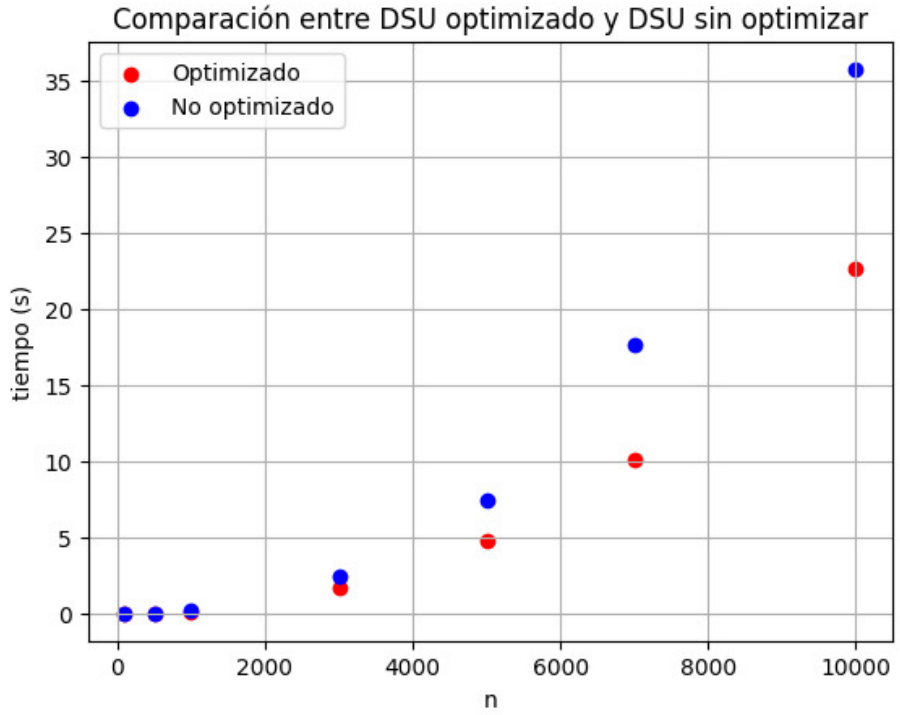
itera la operación *find*. Esto lo logramos haciendo que todos los padres apunten directamente a la raíz. Con esta modificación la complejidad de *find* ya cambia a $O(\log n)$.
Por otra parte, la optimización *Union by rank* nos permite controlar más la velocidad a la que crece la altura del árbol. La idea es definir un rango y unir el árbol de menor rango a la raíz con el mayor rango de todos. Con estas dos optimizaciones la complejidad de peor caso de *union* termina siendo $O(\log n)$, pero con una complejidad amortizada de $O(\alpha(n))$[1].
Ambas optimizaciones por separado, mejoran la complejidad de las operaciones involucradas respectivamente, pero cuando se combinan al mismo tiempo, se logra que tanto *union* como *find*, tengan complejidad amortizada de $O(\alpha(n))$.

Notemos que la cantidad de nodos para los que volvimos a hacer la experimentación es mucho menor que los de la experimentación original. Esto se debe a que en un primer momento nosotros llegábamos a casos como $n = 6.000.000$ porque lo que hacíamos eran suites de tests que constaban de muchos test pequeños, y en total llegaban a sumar esa cantidad de nodos. Para la nueva experimentación nos limitamos a un caso por test.

Para analizar y comparar los tiempos de ejecución de ambos algoritmos, vamos a analizar el peor caso, donde el peor caso para la versión naive de DSU, se da cuando los árboles que se generan crecen linealmente en profundidad. Esto sucede cuando todos los *unite* involucran a un mismo árbol. Para forzar esto lo que hacemos es colocar todas las oficinas en una línea recta de la siguiente forma: la primer oficina la colocamos en la posición 1, la segunda en la 2, la tercera en la 4 y así hasta la n-ésima en la posición $2^{(n-1)}$.

[1] $\alpha(N)$ se refiere a la función inversa de Ackermann la cual crece muy lentamente y para los casos prácticos se puede acotar por $\alpha(N) \leq 4$.

Veamos el gráfico que representa el tiempo contra la cantidad de nodos de ambas versiones del algoritmo, para poder notar más visualmente la diferencia entre ellas.



Podemos ver más claramente como para los casos chicos (entre 100 y 1000 nodos) la diferencia es casi imperceptible, ya que las optimizaciones que le realizamos al DSU, termina haciendo operaciones que no son tan aprovechadas para esta cantidad de nodos.

Ahora, en casos más grandes (entre 3000 y 10000 nodos) podemos observar una diferencia más considerable entre las dos versiones, donde la optimizada lleva la ventaja . Esto se debe a que para estas cantidades de nodos si se le saca provecho al *path compression* y al *union-by-rank*.

Conclusión

Podemos ver cómo nuestras predicciones se ven reflejadas en los datos obtenidos en la experimentación, y concluimos que la versión de Kruskal vista en clase, se va a comportar de forma más eficiente a medida que N vaya creciendo para árboles de gran profundidad.

Esto se debe principalmente a *path compression* ya que gracias a esta optimización se reduce considerablemente la profundidad de este tipo de árboles, y por lo tanto hace que la versión optimizada lleve una ventaja bastante notable frente a la versión naive a medida que N tienda a infinito.

Ahora si hablamos del caso general para ambas implementaciones, la complejidad promedio va a ser similar y por lo tanto no se va a ver una diferencia muy notoria entre ambas versiones. Esto se debe a que para casos promedio uno no fuerza que los árboles generados en el DSU crezcan de forma lineal, y por lo tanto no se le saca mucho partido a *path compression* y *union-by-rank*.